COLONY: A Privileged Trusted Execution Environment with Extensibility

Yubin Xia, *Member, IEEE,* Zhichao Hua, *Member, IEEE,* Yang Yu, Jinyu Gu, *Member, IEEE,* Haibo Chen, *Senior Member, IEEE,* Binyu Zang, *Member, IEEE,* and Haibing Guan, *Member, IEEE*

Abstract—The code base of system software is growing fast, which results in a large number of vulnerabilities: for example, 296 CVEs have been found in Xen hypervisor and 2195 CVEs in Linux kernel. To reduce the reliance on the trust of system software, many researchers try to provide trusted execution environments (TEEs), which can be categorized into two types: non-privileged TEEs and privileged TEEs. Non-privileged TEEs (e.g., Intel SGX) are extensible, but cannot protect security services like virtual machine introspection (VMI) due to the lack of system-level semantics. On the contrary, privileged TEEs (e.g., the secure world of ARM TrustZone) have system-level semantics, but any additional service implemented in the privileged TEE directly increases the TCB of the entire system. In this paper, we propose a new design of TEE to support system-level security services and achieve better extensibility with a small TCB. Each TEE instance of the proposed design is named a COLONY. Specifically, we introduce a *secure monitor* for isolation and capability management. Each COLONY is assigned capabilities to access only necessary system-level semantics. We use the new TEE to build four security services, including secure device accessing, VMI tools, a system call tracer, and a much more complex service to virtualize ARM TrustZone with multiple COLONIES. We have implemented the system on ARMv7 and ARMv8 platforms, in Xen hypervisor and Linux kernel, and perform a detailed evaluation to show its efficiency.

Index Terms—System Security, Trusted Execution Environment, Operating System, Virtualization

1 INTRODUCTION

COMMERCIAL system software (e.g., hypervisor and operating system) contains a variety of system-level security services such as virtual machine introspection (VMI), secure device accessing, system call tracer, kernel integrity checker, random number generator, etc. These services need to be isolated from malicious guest virtual machines (VMs)/processes, and need to access system semantics to provide rich functionalities, so they need to be run in the privileged mode. However, the code base of existing commercial system software is growing fast which results in a large number of vulnerabilities: for example, 296 CVEs have been found in the Xen hypervisor [1] and 2195 CVEs in Linux kernel [2] An exploit of the system software could allow complete access to the entire system and violate the provided security services.

To reduce the reliance on the trust of system software, plenty of researches [11], [15], [22], [25], [28], [33], [44], [45] try to provide trusted execution environments (TEEs) by using a secure monitor to interpose the critical operations of system software, or with the hardware-supported secure enclaves, e.g., Intel's Software Guard eXtension (SGX). These TEEs can be categorized into two types: non-privileged TEEs and privileged TEEs. A non-privileged TEE, such as Intel SGX, supports multiple TEE instances running in the non-privileged mode, which are isolated from each other as well as the system software. Since the code running in a TEE instance is not included in trusted computing base (TCB) of the whole system, new functionalities can be added by launching new TEE instances, and each of them only forms its own disjoint TCB. However, non-privileged TEEs have no system-level semantics, such as memory mapping or I/O processing, which is critical for system-level security services. On the contrary, privileged TEEs are able to control the entire system resources. For example, ARM TrustZone provides a TEE called *secure world* which has the highest privilege. TZ-RKP [12] leverages the secure world for runtime kernel protection. However, existing privileged TEEs can only provide a single privileged TEE instance, which means that adding new security services will increase the TCB of the entire system, which may cause security problems [18].

In order to support the system-level security services and at the same time provide better extensibility, compared to existing solutions, we propose a new TEE that has system-level semantics and does not belong to the system TCB. Each instance of the proposed TEE is named a COLONY. A COLONY has the same privilege mode as the system software, i.e., the hypervisor or the OS. Different COLONIES are isolated from each other as well as the system software. We empower each COLONY to access two types of system-level semantics: *hardware semantics*, including CPU context, exception table, memory mapping and hardware devices, and *software semantics*, which mainly refers to the data of system software in memory. Each COLONY is assigned

• *Y. Xia, Z. Hua, Y. Yu, J. Gu, H. Chen, B. Zang, H. Guan are with the Shanghai Key Laboratory of Scalable Computing and Systems, Shanghai Jiao Tong University.*
Email: {xiayubin, zchua, yuyang, haibochen, byzang, hbguan}@sjtu.edu.cn.
Corresponding author: Zhichao Hua.

Manuscript received Apr 09, 2020.

1. https://www.cvedetails.com/vulnerability-list/vendor_id-6276/XEN.html
2. https://www.cvedetails.com/vulnerability-list/vendor_id-33/Linux.html

capabilities to access these semantics. A secure monitor is introduced to isolate a COLONY from untrusted system software. Two types of isolations are enforced: *data-flow isolation*, which protects a COLONY's data in memory and CPU context, and *control-flow isolation*, which prevents the untrusted system software from bypassing or partially executing a COLONY's code. We also propose interfaces to support COLONY management, including creation, invocation, communication, etc.

To demonstrate the effectiveness and expressiveness of COLONY, we use it to build three system-level security services, including 1) virtual machine introspection (VMI), 2) secure device accessing, and 3) syscall tracing, and present how to use COLONY to fully achieve the security requirements of these services. Then we use multiple COLONIES to implement the virtualization of ARM TrustZone, which provides a virtual TrustZone secure world for each guest, without trusting the hypervisor. This case show how to decouple a complex service into multiple COLONIES.

We implement the system on both HiKey development board (ARMv8) and Samsung's Exynos development board (ARMv7), with Xen hypervisor and Linux kernel as system software. The evaluation results show that our system imposes quite limited performance overhead (less than 2%). The security services inside COLONIES reveal satisfactory performance, too.

In summary, this paper makes these contributions:

- We propose a new design of privileged TEE; each TEE instance, named COLONY, can access system-level semantics and is isolated from other COLONIES.
- We re-design four system-level security services with the help of COLONY: virtual machine introspection, secure device accessing and system call tracing, which are protected with a single COLONY for each, and TrustZone virtualization, which is protected with multiple COLONIES.
- We implement the new TEE design as well as the four case studies, with different system software including Xen hypervisor and Linux kernel, and on different hardware platforms including HiKey (ARMv8) and Samsung's Exynos (ARMv7). A detailed evaluation is performed to show the efficiency of our design.

2 RELATED WORK AND MOTIVATION

There has been a long line of research on constructing trusted execution environments (TEEs) based on various hardware or software. These TEEs can be categorized into two types: privileged TEEs and non-privileged TEEs, according to the privilege level they are running, as shown in the left part of Figure 1. While sharing the same goal: to reduce the reliance on the trust of the system software as much as possible, the two types of TEE have different properties and are suitable for different scenarios.

Non-privileged TEEs: Most existing non-privileged TEEs are built by either using a software secure monitor [21], [28], [39], [45] or directly leveraging hardware-supported secure enclaves [7], [16], [19], [24], [40]. The former one is known as *software TEE*, such as Overshadow [21], CloudVisor [45], Inktag [28], Sego [33] and SICE [13], which provides TEE with different granularities. The latter is the *hardware TEE*, which is enabled by hardware features. Intel SGX [7] protects a piece of user-level code and data encapsulated into *enclaves*. An enclave's memory pages cannot be accessed by other software including the OS and the hypervisor. Haven [15], SCONE [11] and Graphene-SGX [43] further improve the compatibility of the SGX enclave. AMD SEV-SNP [40] encrypts memory with different keys for each VM as well as the hypervisor for isolation and protection. The SEV-SNP provides VMPL0 which could be used to host some secure services, e.g., the secure VMI. However, the VMPL0 mode is non-privileged which cannot access system-level semantics. Furthermore, SEV-SNP cannot be used in non-virtualization environment and is not available on existing hardware yet. SecureBlue++ [16], Sanctum [24] and Bastion [19] also provide non-privileged TEEs based on the hardware design. On ARM platform, OSP [23], TrustICE [42] and Sanctuary [17] leverage virtualization and TrustZone to construct non-privileged TEEs.

Non-privileged TEEs support multiple TEE instances. New functionalities can be added by launching new instances, without increasing the system TCB. One of the major limitations is that non-privileged TEEs do not have system-level semantics, which is required for system-level security services. For example, a kernel integrity checker needs to access kernel's memory; a VMI tool requires to map and access the memory of the target VM. Thus, non-privileged TEEs are not suitable for these services.

Privileged TEEs: Privileged TEEs can be enabled by software, like SKEE [14] and Nested Kernel [26], or by hardware features, like ARM TrustZone for Cortex-A [10]. ARM also provides TrustZone for Cortex-M in IoT scenario, which usually runs simple software stack. In this paper, we only focus on TrustZone-A, and all the "TrustZone" refers to "TrustZone-A". TrustZone splits the CPU into two execution environments: a *normal world* and a *secure world*. In the specification of ARM, all hardware resources can be partitioned into the two worlds. However, some manufacturers may choose to hardwire some devices to the secure world. The secure world is a privileged TEE, which can access all resources. Manufacturers often run TEE OSes [4], [5], [8] in the secure world to construct multiple non-privileged TEEs. However, they cannot access system-level semantics. ARM introduces virtualization extensions in the secure world in ARMv8.4, which can run multiple TEE OSes in the secure world. There is no available hardware yet. Further, the TEE OS still cannot access system-level semantics – only the TEE hypervisor running in sEL2 can directly access all hardware resources and system semantics.

Since privileged TEEs usually can access all the computing resources, they have the system-level semantics and can be used for system-level security services. For example, TZ-RKP [12] implements a runtime kernel integrity protector by leveraging the privileged TEE provided by ARM TrustZone. SKEE [14] and Nested Kernel [26] construct a single privileged TEE by software and use it to implement a runtime kernel monitor. Aurora [36] leverages SMM to protect system services. However, Aurora only enables the service to access devices, and provides a single privileged TEE (called SSV) for all services. Existing privileged TEEs do not support multiple privileged TEE instances, which

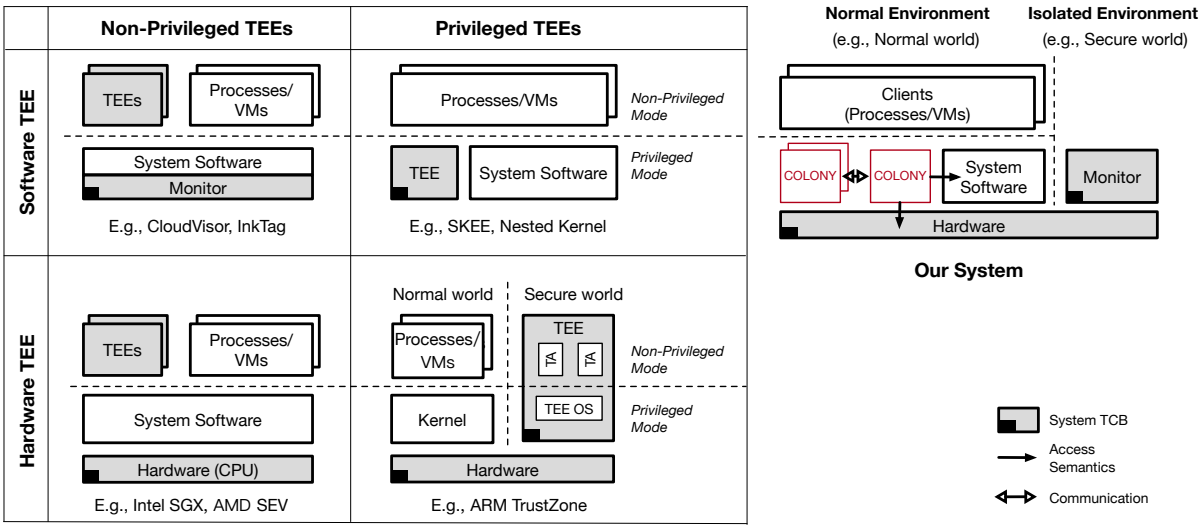


Fig. 1. **Different TEE designs and the overview of our design.** Note that our design can be applied to not only ARM (with TrustZone support), but also other platforms, as long as it provides isolated environment for the secure monitor.

means that adding new security services will increase the TCB of the entire system.

Motivation: There have already been a variety of system-level security services. VMI helps the user to monitor the runtime status of a guest VM; syscall tracer allows a parent process to monitor the execution of its child; disk encryption transparently protects the confidentiality of on-disk data. Most of these services rely on the system-level semantics to achieve rich functionalities and minimal performance overhead. Many of the services run at low level of the system and have little reliance on the functionalities of the system software. By running these services in TEE, we can protect them from untrusted system software.

Comparing with putting multiple services into one TEE instance, it is more secure to deploy them into multiple small instances, and let them interact through explicit communication channels [31]. Recent work [18] also shows that if putting too much logic into a single TEE instance (e.g., the secure world of TrustZone), even commercial implementations may contain critical vulnerabilities. Thus, it is required to construct multiple privileged TEE instances, and each instance should be able to access system-level semantics and not be a part of the system TCB.

3 COLONY ABSTRACTION

Our system has following three goals: 1) **Isolation**: a COLONY should be isolated from the untrusted system software and other COLONIES. 2) **System-level Semantics**: a COLONY should have the ability to access system-level semantics. 3) **Minimal TCB**: there are two kinds of TCBs: system TCB and service TCB. The system TCB includes all the components that support the functionalities and the isolation of COLONY. Components outside the system TCB cannot break the above two security goals. On the contrary, the service TCB depends on the semantic of service. It contains components that are trusted by other parts of the service, which may affect the whole service once compromised by attackers. All COLONIES should be excluded from the system TCB so that any compromised COLONY cannot break the isolation between different COLONIES or provide fake system-level semantics for others. For a complex service, we allow it to be decoupled and to be deployed in multiple COLONIES with different capabilities, to reduce the service TCB. Section 7.1 discusses more about these two TCBs. In this section, we will first introduce the threat model of our system, and then give the COLONY abstraction, followed by how it accesses system-level semantics.

3.1 Threat Model

We have following assumptions: 1) the hardware components are trustworthy; 2) the system supports an execution environment that is isolated from the system software and can access all hardware resources (e.g., the secure world of TrustZone or a tiny hypervisor), which is used to run a secure monitor. 3) the secure boot is used to guarantee the integrity of the system during the boot-up process, but not the integrity thereafter; 4) the system software, is benign but has bugs, which will finish the initialization correctly during booting, but may get fully compromised after that; 5) any user application or guest VM may be malicious; 6) one COLONY could be compromised and issue attacks to other COLONIES.

Our system focuses on protecting system services, and leaves the protection of user applications or VMs to existing user-level TEEs, which is orthogonal to our work. For example, in the scenario of one process using some system services, we assume that the process has been protected by some user-level TEE like TZ-Container [30]. Even so, the application still relies on the correctness of the system services, which will be protected by COLONY.

This paper does not consider the case that a system-level service itself leaks its own data. DoS attacks, side-channel attacks and physical attacks are also out of the scope.

3.2 Overview of COLONY

As shown in Figure 1, a COLONY runs in the same mode with the system software, either the hypervisor or the

TABLE 1
The interfaces of a COLONY.

Type	Name	Arguments	Description
Management interfaces	CREATE	Execution info, interface info	Create a COLONY and get the ID.
	ASSIGN_CAP	ID, capability type, permission, info	Assign a capability to a COLONY.
	ALLOW_COMM	Source ID, target ID	Specify the target a COLONY can communicate with
Invocation and communication interfaces	INVOKE	ID, arguments	Allow the client to invoke a COLONY.
	Invocation Memory	None	Allow to invoke a COLONY by accessing specific memory.
	COLONY_CALL	ID, arguments	Allow a COLONY to call another COLONY.
	COLONY_RETURN	return value	Allow a callee to return to the caller COLONY.
Runtime interfaces	ALLOC PAGES	Memory type, number	Allocate pages from COLONY memory.
	GET_SEMANTICS	Semantics type, metadata	Get a memory pointer used to access the target semantics.
	WRITE_SEMANTICS	Semantics type, metadata, data	Modify the target semantics.
	ENTER_COLONY	None	Enter a COLONY.
	EXIT_COLONY	None	Exit current COLONY.

TABLE 2
The capability types of a COLONY.

Capability Type	Description
CPU context	Which parts of CPU context can be read/written by a COLONY.
Exception table	Whether a COLONY can read/write the exception table.
Memory mapping	Whether a COLONY can read/write/check the memory mappings.
Hardware device	Which device can be accessed by a COLONY.
System memory	Whether a COLONY can read/write the memory of system software.

OS kernel. We use a monitor, protected by either a tiny hypervisor or the secure world of TrustZone, to construct COLONIES. A COLONY can host a system-level service to serve multiple clients. The client can be a guest VM or a user process. Both the **data-flow** and the **control-flow** of a COLONY are isolated from the untrusted system software as well as other COLONIES. Each COLONY has its own code and data region to support a system-level security service. There are two kinds of data regions: **user-private memory** and **global memory**. We maintain a user-private memory region for a *client*, to store the service data for the client. When handling a request, the COLONY can only access the user-private memory of current client. The global memory is used to store the service data shared by all clients.

Our system provides a capability method for COLONIES to access system-level semantics. The developer can give each COLONY the minimal capability. The capability method also enables a complex service to be decoupled and run in multiple COLONIES for stronger isolation. We also provide three kinds of interfaces of COLONIES: management interfaces, invocation/communication interfaces, and runtime interfaces, as shown in Table 1.

3.3 Capability for Accessing System-level Semantics

We divide system-level semantics into two types: **hardware semantics** and **software semantics**. The previous one consists of CPU context, exception table, memory mapping and devices. The only software semantic we provide is system memory. Table 2 shows all the five capabilities.

Each type of capability corresponds to a kind of semantics, and each of them has different permissions: read/write/check/access. All the capabilities have both the *read* and *write* permissions, except the hardware device which only has *access* permission. For the memory mapping, we further provide a *check* permission to indicate whether a COLONY can check mapping modifications. For the CPU context, the permission is maintained for each CPU register.

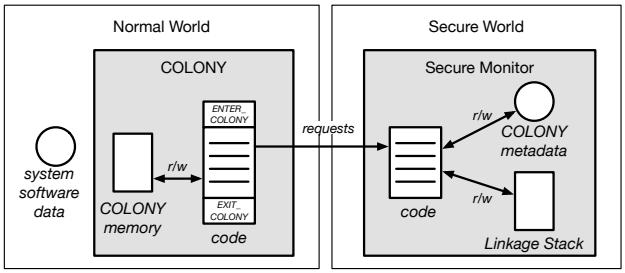


Fig. 2. **Architecture of our system**.

4 Design and Implementation

The abstraction of COLONY is portable and can be implemented on either x86 or ARM platform, with a secure monitor protected by, e.g., a tiny hypervisor or ARM TrustZone. In this section, we propose a detailed design based on ARM TrustZone. The design overview is shown in Figure 2. We first introduce the address translation on ARM and further give the design of our system.

4.1 Address Translation on ARM

COLONY's memory isolation employs basic operations of memory management unit (MMU) [3]. With MMU, the address translation is performed with translation tables, which is pointed to by a translation table base register. There are two different kinds of address translations: one-stage and two-stage. The previous one maps a virtual address (VA)

3. This paper focuses on the Cortex-A platform which is widely used to run complex OS and hypervisor. Other ARM profiles, e.g., Cortex-M, may not provide MMU.

to a physical address (PA). It is used for the OS kernel (no virtualization) or for the hypervisor. The page table is controlled by the OS or the hypervisor respectively. The two-stage translation is used by guest VMs, which has two stages: in stage-1, a VA is translated to an intermediate physical address (IPA); in stage-2, the IPA is further translated to a PA. The stage-1 page table is controlled by guest OS and the stage-2 page table is controlled by the hypervisor.

4.2 Design Challenges

There are three goals of our system: strong isolation, system-level semantics and minimal TCB. Each of them derives multiple challenges.

Challenge-1: isolating a COLONY. A compromised system software running in privileged mode may: 1) modifies the memory and CPU context of a COLONY, 2) hijacks a COLONY's execution, partially execute a COLONY or totally bypass it, or 3) indirectly replaces the arguments passed to it. Thus, both the **control-flow** and **data-flow** of a COLONY should be isolated.

Challenge-2: accessing system-level semantics. We allow each COLONY to access system-level semantics. The system software may try to tamper with the semantics or provide fake semantics.

Challenge-3: minimizing the TCB. Both the system TCB and service TCB should be minimized. Since all COLONIES run in the privileged mode, any compromised one may tamper with the system semantics and further compromise others.

4.3 Isolating Data-Flow of COLONY

To protect the data-flow of a COLONY, the secure monitor isolates both the CPU context and memory.

To isolate the CPU context, the monitor maintains a copy of the context for each COLONY, including CPU registers. Whenever entering or exiting a COLONY, the monitor switches the context. Similar to the syscall, COLONY invocation uses registers to transfer the arguments and return value, which will not be switched. To hook the entering and exiting operations, we implant *ENTER*_COLONY and *EXIT*_COLONY interfaces to the entry and exit point of a COLONY, as shown in Figure 2. These two interfaces will invoke the secure monitor.

To isolate the memory, the monitor constructs different mappings for COLONIES and system software. If controls the memory mappings exclusively and traces the mappings of each physical page. Thereafter, the monitor enforces that a COLONY's memory can never be mapped to the system software or other COLONIES. The monitor switches the translation table when entering or exiting a COLONY. The switching is only performed on current CPU core.

The monitor needs to exclusively control all the mappings to the physical memory in the normal world, including all VA-to-PA mappings, as well as the IPA-to-PA mappings. The monitor deprives system software of the ability to modify memory mappings, so that they must invoke the monitor to load a translation table or to modify table entries. We use OS kernel as an example: there are three ways for a kernel to modify memory mappings: 1) changing the base register to a new translation table, 2) modifying the table entries, or 3) disabling the address translation [4]. We modify the kernel to ensure that there is no MMU-related instruction which loads a new translation table or disables the translation. After that, the monitor maps translation tables as read-only to the kernel. All mapping modification operations are replaced with invocations to the monitor.

The OS kernel may try to bypass the monitor by: 1) injecting MMU-related instructions to the kernel code; 2) leveraging ROP to form these instructions; or 3) jumping to the user process to execute these instructions. To prevent the first way, the monitor maps the kernel's code pages as read-only. There are two situations that the kernel needs to add new executable pages: swapping and loading kernel modules. The monitor checks the integrity of the memory page during the swapping. All kernel modules are checked by the monitor to ensure that they do not contain these instructions. To prevent ROP, we remove all the ROP gadgets which can be used to form new MMU-related instructions. It is relatively trivial on ARM platform which has fixed-length ISA. We also consider all the ARM ISAs with different lengths. To prevent return-to-user attacks, the monitor maps user pages as non-executable for the kernel. The above method can also be used for the hypervisor.

Cross-Client Isolation of COLONIES: Besides isolating a COLONY from the system software and other COLONIES, the monitor also provides cross-client isolation. A COLONY has multiple user-private memory regions, and each of them is used to store per-client data. When a COLONY is invoked, the monitor identifies current client (by the VMID of a VM or the PCID of a process), and maps the corresponding user-private memory. For the first time a COLONY is invoked, the monitor will allocate the user-private memory.

4.4 Isolating Control-Flow of COLONIES

The monitor isolates the control-flow of a COLONY by 1) isolating the control-flow data from other COLONIES and the system software, 2) preventing a COLONY from being partly executed, and 3) preventing a COLONY from being bypassed. The control-flow data is protected by data-flow isolation, so we will focus on the latter two points.

To prevent the system software from executing part of a COLONY, the monitor enforces the atomicity: a COLONY can only execute from its entry point, and must run to completion. For the previous goal, the monitor unmaps a COLONY's code from the system software, except the started *ENTER*_COLONY function which is in a different executable page. Thereafter, a COLONY can only be invoked by executing its started *ENTER*_COLONY. The monitor then switches to the COLONY's page table on current CPU core. To enforce the atomicity, the monitor disables interrupts during the execution. We also require that the COLONY's code is self-contained.

To prevent the system software from bypassing a COLONY, the monitor enforces the non-bypassability: once a COLONY is invoked, it must be executed thereafter. Unlike atomicity, non-bypassibility can defend against control flow hijacking attacks, e.g., by completely redirecting the control

4. ARM provides multiple methods to disable the translation table, including modifying the SCTLR_EL1 or TCR_EL1 registers.

flow to other components. Our system provides two invocation interfaces for the client, both are implemented with exceptions (more details in Section 4.6). The monitor forbids the system software from controlling the exception handlers and hooks all the handlers to enforce the non-bypassability.

There are three ways for the system software to control an exception handler: 1) modifying the code of a handler, 2) modifying the exception table, and 3) modifying the exception table base register to enable a new table. To prevent all these methods, we first replace the instruction, used to modify the exception table base register from the system software, with invocations to the monitor. Thereafter, the monitor maps the exception table as read-only to the system software. All the exception handlers are part of the system software's code, which is also mapped as read-only by the monitor. After exclusively controlling the exception handler, the monitor implants a hook in each handler.

4.5 Accessing System-Level Semantics

A COLONY can access five kinds of system-level semantics: CPU context, exception table, memory mapping, hardware devices and system memory. Our system ensures that the semantics are not faked, and provides the capability mechanism to control the access.

CPU Context: A COLONY can access the CPU context of the system software or other clients. To provide a real context, the monitor hooks all the switchings between different privileged modes, identifies current client and saves the CPU contexts.

To switch from low privilege to high privilege, an exception must be triggered The monitor hooks all exception handlers, and can further save the CPU context. Current client can be identified by the VMID/PCID in the CPU context. To switch from high privilege to low privilege, system software must execute specific instructions (e.g., *eret*). We replace all these instructions with invocations to the monitor.

When a COLONY wants to access the CPU context of the system software or a client, the monitor will find the target context and maps it to the COLONY. The system software cannot provide fake context to a COLONY.

We provide two permissions, *read* and *write*, for each CPU register for access control. For read operations, the monitor maps the CPU context to the COLONY. Write operations can be done by invoking the monitor.

Exception Table and Memory Mapping: The monitor has exclusively control of the exception table and translation tables. We provide two permissions, *read* and *write*, to access these two semantics. For the one with read permission, the monitor maps the semantics to a COLONY's address space. The modification can be performed by invoking the monitor.

For the memory mapping, the monitor further provides *check* permission that allows a COLONY to update its mapping policies. A policy is specified by a triple *(REGION_1, OP, REGION_2)*. A memory region includes a virtual memory range and an address space ID. Our system provides two kinds of *OP*: 1) two regions cannot overlap, 2) two regions must be mapped to the same physical region. When the monitor receives a mapping modification request, it enforces that the request satisfies all the policies.

Hardware Device: On ARM platforms, all devices are controlled by accessing corresponding MMIO. The monitor allows a COLONY to directly access a device by mapping its MMIO region to the COLONY. For each device, we provide two kinds of *access* permissions: *sharing access* and *exclusive access*.

Different COLONIES with the *sharing access* permission are allowed to access a device at the same time. When the device is used by a COLONY, the monitor still needs to forbid system software from tampering with the device status. When entering a COLONY with *sharing access* permission of a device, the monitor will unmap the device's MMIO region from the system software. To perform this operation, the monitor traces all the mappings of the MMIO region for each device. When the COLONY exits, the monitor will remap the MMIO region to the system software.

A COLONY with the *exclusive access* permission can access the corresponding device exclusively. No matter whether this COLONY is invoked or not, the monitor will unmap the MMIO region of the device from the system software and other COLONIES, even when they have *sharing access* permission. For each device, the *exclusive access* can only be assigned to one COLONY.

System Memory: A COLONY is allowed to map and access the memory of the system software to implement services such as kernel integrity monitor. The system memory includes code, data, exception table and translation tables. We provide two permissions, *read* and *write*, to control the access of the system memory. For a COLONY with write permission, only the system software data is mapped with write permission. The code, exception table and translation tables are mapped as read-only.

4.6 COLONY Interfaces

Invocation Interfaces: To enforce the non-bypassability of COLONY, we provide *INVOKE* and *invocation memory* for a client to invoke a COLONY. Both interfaces are non-bypassable. The previous one is implemented as a new syscall/hypercall in OS kernel/hypervisor, and the latter works similar as the MMIO of a device. On ARM platform, both the syscall and the hypercall are implemented with exception. For the *invocation memory*, accessing it also triggers an exception. As mentioned in Section 4.4, the monitor has hooked all the exception handlers. Thus, the monitor can detect the invocation of the two interfaces and invoke the target COLONY, which enforces the non-bypassability of the interfaces. The protection of the arguments and return value is also performed by the monitor.

Cross-COLONY Communication Interfaces: The COLONY_*CALL* and COLONY_*RETURN* are used for cross-COLONY communication. As shown in Figure 2, the monitor keeps a **linkage stack** for each CPU core. The stack is used to maintain the invocation context. Each entry of the stack is called **linkage record** which consists of the caller COLONY ID, callee COLONY ID and return address. For a call operation, the monitor pushes a new record to the stack, switches to the callee's address space and starts its execution. The monitor also checks whether the callee is a legal target (specified through the *ALLOW_COMM* interface). For a return operation, the monitor pops the

record from the linkage stack and switches to the caller. The arguments and return value are transferred through the CPU register with the help of the monitor. The monitor also passes the caller ID as an argument to the callee.

Management Interfaces: The *CREATE* interface is used to create a new COLONY, which can only be performed during system boot, before the first process/VM (init process for Linux and Domain-0 for Xen) is created. The monitor handles the *CREATE* interface.

After creating a COLONY, the system software can assign capabilities to the COLONY or specifies its legal communication target. The *ASSIGN_CAP* configures the capability and takes four arguments: 1) the ID of the target COLONY, 2) the capability type, 3) the permission, and 4) some auxiliary information. For example, the invocation of *ASSIGN_CAP(RTIC_COLONY, DEVICE, ACCESS, RTIC_INFO)* will assign the capability of RTIC device to the RTIC_COLONY. The *ALLOW_COMM* specifies which COLONY can be invoked. The invocation of *ALLOW_COMM(COLONY-1, COLONY-2)* means COLONY-1 can call COLONY-2.

Runtime Interfaces: Each COLONY can invoke these interfaces at runtime to access system-level semantics. Table 1 shows all five runtime interfaces. For example, a COLONY can invoke *GET_SEMANTICS(DEVICE, RTIC_INFO)* to access RTIC device. If it has *ACCESS* permission of RTIC, the monitor will return a pointer which points to the MMIO region of RTIC with both read and write permissions.

5 CASE STUDIES USING COLONY

To show the expressiveness of the COLONY abstraction, we use it to protect four system-level security services, including virtual machine introspection (VMI), secure device access, syscall tracer and TrustZone virtualization.

For each service, we first analyze and summarize the required security properties before the redesign, as Table 3 shows. The first property **P-0** is a general one that is enforced by locating the service inside a COLONY. Other properties will be introduced along with each service.

5.1 Case Study I: Virtual Machine Introspection

Motivation: Virtual machine introspection (VMI) is widely used in virtualization environment. Even when the hypervisor is untrusted, the VMI can still help a manager VM monitor the status of another functional VM. Although both VMs are protected by existing non-privileged TEEs, we still need to protect the VMI service from untrusted hypervisor.

Threat Model: We assume that the hypervisor is untrusted, and VMs are protected by existing user-level TEEs. Besides directly tampering with the status of the VMI service, the untrusted hypervisor can also bypass the VMI (**P-1.1**) by not triggering its procedure at user-defined position, or let the VMI service access fake memory or CPU context of the target VM (**P-1.2 and P-1.3**).

Design and Implementation: Existing VMI tools (e.g., libvmi [38]) of Xen run in the user space of Domain-0. Each time to access a page of the target VM, libvmi needs to ask Xen to map the physical memory to Domain-0 and then relies on the kernel of Domain-0 to map the memory to libvmi's virtual address space. We run the secure VMI inside a COLONY to prevent the malicious Xen from directly tampering with the service status. Leveraging the capability of accessing system-level semantics, the secure VMI within a COLONY can directly access the real memory mappings and CPU context of the target VM, which enforces **P-1.2** and **P-1.3**. We use *INVOKE* interface to allow the VM invokes the secure VMI, so that the service cannot be bypassed by the untrusted hypervisor (**P-1.1**). Currently, we implemented four basic functionalities of VMI: reading kernel symbols, reading kernel addresses, dumping task list and dumping kernel module list.

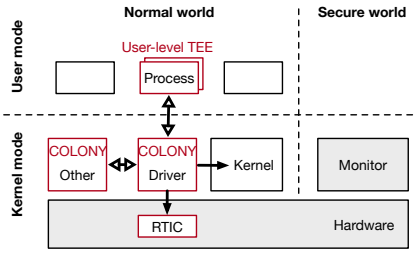


Fig. 3. **Secure device accessing based on COLONIES.**

5.2 Case Study II: Secure Device Accessing

Motivation: There exist many devices which provide security related functionalities, also known as secure devices. User processes require these hardware devices to implement some secure functionalities. For example, the Runtime Integrity Checker (RTIC) can help calculate the hash value of a specific memory region. The secure kernel mode of TrustZone can be used to implement a secure device accessing system. However, managing all secure devices in such single privileged TEE will significantly increase the system TCB. In this case, we try to use a COLONY to enable the secure accessing of a device.

Threat Model: We assume that the hardware implementation is correct, but the kernel could be compromised. The device access service can be protected within a COLONY. All other COLONIES may be malicious. We assume that user processes are protected by existing TEE methods. However, the untrusted kernel could directly leak or tamper with the device status (**P-0**), hijack the interfaces between the process and the device access service (**P-2.1**), or help a malicious process access other's device status (**P-2.2**).

Design and Implementation: We use a COLONY to protect the logic for accessing the device, as shown in Figure 3. The functionality of the service is similar as a device driver, which manages the hardware status and enables processes to access the hardware. We use the *INVOKE* API to allow processes to invoke the service securely, which enforces **P-2.1**. The device accessing service manages the device status for different processes. We store each process's status in the user-private memory of the COLONY, so that one process cannot access other's status, which enforces **P-2.2**. We choose RTIC as an example to show how to use the above design to enable secure device accessing for user processes. We give a COLONY the ability to access the RTIC exclusively. After that, this COLONY handles the RTIC accessing request from user processes and performs the operation on the real hardware.

TABLE 3
Required properties of security services.

Security Services	Security Properties	Properties Violation by System Software → Consequence
General Service	**P-0**: Control flow and data flow of a service should be isolated.	Tamper with the control flow or data flow. → Controlling the execution of the service.
Virtual Machine Introspection	**P-1.1**: VMI must be performed at specific points. **P-1.2**: VMI accesses the real memory of a guest. **P-1.3**: VMI accesses the real CPU context of a guest.	Not perform VMI. → Bypassing the protection of VMI. Providing fake memory. → Hiding guest's wrong status. Providing fake CPU context. → Hiding guest's wrong status.
Secure Device Access	**P-2.1**: Integrity of input/output should be protected. **P-2.2**: Each process can only use its own device.	Tamper with the input and output. → Providing malicious device. Let a process access other's device. → Info leakage/data corruption.
Syscall Tracer	**P-3.1**: Integrity of tracing result should be protected. **P-3.2**: Only trace real syscall. **P-3.3**: All syscalls can be traced.	Tamper with the result. → Hiding malicious operation. Let tracer trace fake syscall. → Injecting arbitrary syscall. Let tracer trace part of syscall. → Hiding malicious syscall.

5.3 Case Study III: Syscall Tracer

Motivation: Syscall tracer, e.g., the *ptrace* provided by Linux kernel, allows a parent process to trace all the syscalls invoked by its child process. It can be used to detect the malicious syscall invocation of the child process. For example, a process may be compromised by an attacker, even when the process is protected within a user-level TEE. After that, the attacker could invoke some syscalls to perform malicious behaviors. Secure syscall tracer can detect these illegal invocations without trusting the OS. We try to protect the tracer with a COLONY.

Threat Model: We assume that the OS kernel is untrusted, and user processes are protected by existing user-level TEEs. The malicious kernel may directly tamper with the tracing result (**P-3.1**). It may also fake the tracer to trace fake syscall (**P-3.2**), or hide some syscall invocations (**P-3.3**).

Design and Implementation: We run the secure syscall tracer inside a COLONY. On ARM, the syscall is implemented with exception, and the secure monitor has already hooked all exception handlers. Based on that, the tracer can be implanted at the syscall entry to hook all syscalls invoked by user processes (**P-3.1**). Because of the isolation mechanism of our system, the compromised kernel cannot inject fake syscall to the tracer (**P-3.2**). A user process can use *INVOKE* interface to invoke the secure tracer, which also enforces the integrity of the tracing result (**P-3.1**).

5.4 Case Study IV: Virtualizing TrustZone

Complex services can be decoupled and deployed with multiple COLONIES. For demonstration, we design and implement a new service called TrustZone virtualization (vTZ).

Motivation: TrustZone has already been widely used to build secure system [12], [32], [34], [35], [41]. However, current TrustZone hardware only provides one secure world, which is not enough for multiple guest VMs in virtualization environment [5]. Although existing systems, such as Sanctuary [17], could construct multiple user-level TEEs on ARM, cloud users cannot directly run their TrustZone-based systems within these TEEs. The goal of vTZ is providing a virtual secure world for each guest, and enforcing that the virtual secure world has the same functionalities and security properties as the real hardware.

5. The lastest design of TrustZone supports hardware virtualization. Here we virtualize TrustZone to show the expressiveness of our TEE.

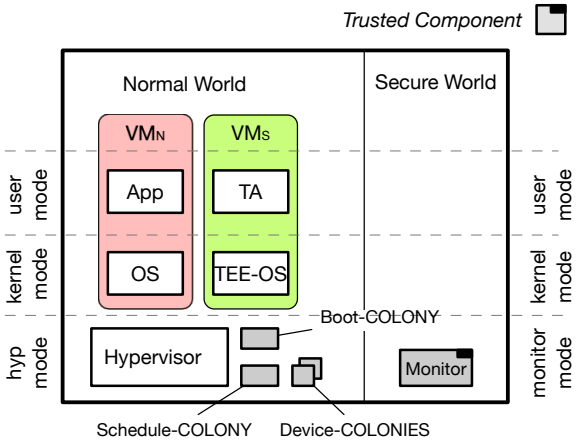


Fig. 4. **TrustZone virtualization**. VM_n and VM_s mean the normal world and secure world of a VM, respectively.

Threat Model: vTZ relies on the normal world hypervisor to virtualize functionality of guest secure worlds and uses the COLONIES to enforce protection. As shown in Figure 4, the two virtualized worlds of each guest (guest normal world and guest secure world) are simulated by two different VMs (called VM_n and VM_s). Then we analyze a set of security properties that physical TrustZone provides, as shown in Table 4, and leverage multiple COLONIES to enforce all these properties. We assume that the hypervisor and the host OS are untrusted, and any other users may be malicious. The attacker may try to break any property mentioned in the Table. vTZ also assumes that the secure boot is used to protect the integrity of the hypervisor's code, during the system boot.

Design and Implementation: We will further give the detailed implementation of vTZ, including how to virtualize the secure boot, protect CPU states and virtualize the resource partitioning.

5.4.1 Virtualizing Secure Boot

Secure boot is used to ensure the integrity of booting. The booting process of a TrustZone-enabled device contains following steps: 1) Loading a bootloader from ROM, which is tamper resistant. 2) The bootloader initializes the secure world and loads a secure OS to memory. 3) The secure OS initializes the secure world. 4) The secure OS switches to the

TABLE 4
Properties enforced by TrustZone which should also be enforced by vTZ.

TrustZone Features	System Properties	Properties Violation by Malicious Hypervisor → Consequence
Secure Boot	**P-4.1.1**: *S/W must boot before N/W.* **P-4.1.2**: *Boot image of S/W must be checked.* **P-4.1.3**: *S/W cannot be replaced.*	Violate boot order. → Secure configuration bypass. Violate integrity check of boot image. → Code injection in guest S/W. Replace a guest S/W with another one. → Providing malicious S/W.
CPU States Protection	**P-4.2.1**: *smc must switch to the correct world.* **P-4.2.2**: *Protect the integrity of N/W CPU states during switching.* **P-4.2.3**: *Protect S/W CPU states.*	Switch to a wrong guest S/W. → Providing malicious S/W. Tamper with CPU states during switching. → Controlling execution of guest S/W. Tamper with guest S/W's CPU states. → Controlling execution of guest S/W.
Memory Isolation	**P-4.3.1**: *Only S/W can access secure memory.* **P-4.3.2**: *Only S/W can configure memory partition.*	Let arbitrary VM access guest secure memory. → Info leakage. Let arbitrary VM configure guest memory partition. → Reconfigure secure memory as normal.
Peripheral Assignment	**P-4.4.1**: *Secure interrupts must be injected into S/W.* **P-4.4.2**: *N/W cannot access secure peripherals.* **P-4.4.3**: *Secure peripherals are trusted for S/W.* **P-4.4.4**: *Only S/W can partition interrupt/peripherals.*	Forbid interrupt being injected into guest S/W. → Disturbing the execution of guest S/W. Let guest N/W access secure peripherals. → Info leakage of secure peripherals. Provide malicious peripherals for guest S/W. → Info leakage of guest S/W. Let arbitrary VM configure guest interrupt/peripherals. → Reconfigure secure peripheral as normal.

normal world and executes a kernel-loader. 5) The kernel-loader loads a non-secure OS and runs it.

In these steps, each time a loader loads a binary image, it will calculate the checksum of the image to verify its integrity. At the same time, the booting order is also fixed: the secure OS is the first to run so that it can initialize the platform first. To virtualize the secure boot process, we need to enforce **P-4.1.1**, **P-4.1.2** and **P-4.1.3**.

During system booting, the hypervisor initializes the data structure and allocates memory for both guest VM_n and VM_s, loads the secure OS image and guest normal world OS image to the memory, respectively. Then the hypervisor needs to register the two VMs in the *boot* COLONY. Thereafter, leveraging the non-bypassability of COLONY, the *schedule* COLONY checks all switchings between the hypervisor and a VM and ensures that only registered VM can be executed. The scheduling of VMs is done by the hypervisor, and the schedule COLONY enforces that the VM_s must run before the corresponding VM_n (**P-4.1.1**).

During registration, the boot COLONY first removes all the mappings of memory pages allocated to the guest from the hypervisor's translation table and checks the integrity of the image of the guest secure OS. Then it creates a binding between the guest VM_n and VM_s by recording their VMIDs, and marks their context data as read-only to hypervisor. The boot COLONY also initializes the stage-2 translation tables of these two VMs and set the VMID in the stage-2 translation table base register. Leveraging the data-flow and control-flow isolation of COLONY, all these operations cannot be influenced by the hypervisor. And all the modifications on memory mappings are checked by the secure monitor. So the **P-4.1.2** and **P-4.1.3** are enforced.

5.4.2 Protecting CPU states

vTZ needs to enforce **P-4.2.1**, **P-4.2.2** and **P-4.2.3** to provide the same CPU states protection of TrustZone. The schedule COLONY intercepts all the switchings between a guest VM and the hypervisor.TrustZone can split hardware resources to A switching includes saving states of the current VM, finding the next VM, and restoring its states. The states saving and restoring are done by the schedule COLONY, while finding the next VM is done by the hypervisor. Then schedule COLONY checks the restored VM to ensure that **P-4.2.1** and **P-4.2.2** are satisfied. During execution, it also prevents the hypervisor from stealing or tampering with VM_s's context to achieve **P-4.2.3**. For example, if one VM exits because of the scheduling, then its CPU states cannot be modified. Further, VM_s's system control registers also cannot be modified by the hypervisor.

5.4.3 Virtualizing Resource Partitioning

TrustZone can assign hardware resources to the normal world or the secure world. Three different resource partitions are provided, together with three different controllers which are used to configure the partition: 1) **Memory partitioning** configured by TrustZone Address Space Controller (*TZASC*); 2) **Peripheral partitioning** configured by TrustZone Protection Controller (*TZPC*); 3) **Interrupt partitioning** configured by General Interrupt Controller (*GIC*).

Once set as secure, the resource can only be accessed by the secure world. A secure interrupt must be injected into the secure world and will lead to a world switching if it happens in the normal world. The secure world can use these controllers to repartition the resource. We provide the same functionality of resource partitioning as a real TrustZone, which includes the configuration of partitioning and the enforcement of partitioning.

Virtualizing Partitioning Controllers: **P-4.3.2** and **P-4.4.4** should be satisfied for virtualizing partitioning controllers. The virtualization of partitioning controllers is done by the "trap and emulate" method. vTZ leverages COLONIES to implement three virtual controllers (*vTZASC*, *vTZPC* and *vGIC*) for each guest. COLONY ensures that all three virtual controllers cannot be compromised by the untrusted hypervisor. Since ARM only provides memory mapped I/O, all these COLONIES use *invocation memory* to interact with guest VM. These virtual controllers check

whether the trap is raised by a VM_s before doing the repartition, which enforces **P-4.3.2** and **P-4.4.4**.

Secure Memory Partitioning: Secure property **P-4.3.1** is a fundamental one. With the help of the secure monitor, the virtual memory partitioning controller (i.e. vTZASC) can enforce the following policy: any guest's secure memory region can only be mapped in its VM_s but not any other VMs or the hypervisor.

Secure Device Partitioning: In secure device part, we must enforce **P-4.4.2** and **P-4.4.3**. We emulated commonly used secure devices within COLONIES, e.g., *TZASC*, *TZPC*, *GIC*, *uart* and *random number generator*. If the device is marked as a secure peripheral by one guest, the emulator will enforce that it cannot be accessed by this guest's VM_n, so the **P-4.4.2** is enforced. One guest's configuration will not influence others. The COLONIES can enforce that all the operations on virtual secure device will eventually be handled by itself, and **P-4.4.3** is satisfied.

Secure Interrupt Partitioning: For securely partitioning interrupts, we need to ensure **P-4.4.1**. We implemented an interrupt dispatcher in a COLONY, which hooks all interrupt handlers and decides whether an interrupt is secure or not according to a virtual interrupt partition list which is managed by the virtual GIC, thus **P-4.4.1** is enforced.

6 PERFORMANCE EVALUATION

We try to answer three questions for evaluation:

- *Question-1*: How does our design influence the performance of the hypervisor and the Linux?
- *Question-2*: How is the performance of a service running in a COLONY?
- *Question-3*: How is the performance of a complex service running in multiple COLONIES?

6.1 Evaluation Environment

We evaluate the performance of our design on both a HiKey ARMv8 board with Kirin 620 SoC (64-bit) and an Exynos Cortex ARMv7 board (32-bit). The HiKey board enables eight 1.2 GHz cores and 2GB memory. The Exynos board enables one 1.7 GHz core and 1GB memory. In virtualization environment, we use Xen 4.4 [9] as the hypervisor and Linux 4.1 as the guest kernel and Domain-0 kernel. On the Exynos board, each guest together with Domain-0 has one virtual CPU. On the HiKey board, each guest, as well as Domain-0, has one virtual CPU and each virtual CPU is pinned on a physical core. In native environment, we use Linux 4.1 as the kernel. The ARM's performance monitor unit (PMU) is used to measure the clock cycles.

On HiKey, we implement the secure monitor as a service inside ARM Trusted Firmware [2]; and on Exynos, the monitor is integrated with the secure boot-loader provided by Samsung. Other functionalities inside the EL3, e.g., the power services (PSCI), are orthogonal to the monitor. For the system software, we perform two modifications: 1) replacing all sensitive instructions related to MMU and exception, with invocations to the monitor; and 2) invoking the monitor to modify page tables and exception tables. Thanks for the well programming of Linux and Xen, almost all modifications are performed on limited inline functions.

TABLE 5
Results of LMBench (in μs).

Operation	Linux	Linux-COLONY
Null syscall	0.22	0.57
Null I/O	0.71	1.03
Open/Close	7.48	8.57
Mmap	252	404
Page Fault	1.24	2.53
Signal Handle	0.65	1.01
Fork syscall	2020	2252
Exec syscall	3022	3316
ctxsw 2p/0k	8.82	14.1

6.2 Overhead of Our Design

LMBench [37]: LMBench contains a series of micro-benchmarks to measure individual OS operations. We use it to measure the overhead of the critical operations of OS (e.g., syscalls and signal handling) on HiKey board, to answer *Question-1*. We run LMBench tool ten times and the average result is shown in Table 5. The Null syscall and the ctxsw (context switch) shows the overhead caused by hooking all the switchings between user and kernel. Fork and exec syscall, as well as the page fault, show overhead of hooking all modifications of memory mapping. Although there is some overhead on part of these operations, it will not dramatically influence the performance of real applications.

SPEC_CPU 2006 [27]: We evaluate *all* SPEC_CPU 2006 INT applications under both the virtualization environment and native Linux on HiKey board, to show the overhead of our system. For each case, we run the benchmark tool ten times. Figure 5(a) and (b) show the normalized overhead of our system on Xen and Linux, respectively. We can see that enabling COLONIES in both Xen and Linux increases very small overhead, about 1-2% on average.

6.3 Security Services With Single COLONY

To answer *Question-2*, we measure the performance of three security services based on COLONIES: 1) secure virtual machine introspection (VMI); 2) secure device accessing and 3) secure syscall tracer. The first two are implemented in COLONIES of Xen and the last is in COLONIES of Linux.

6.3.1 Virtual Machine Introspection

We evaluate both the libvmi [38] and secure VMI with COLONY. We test three basic functionalities: 1) dumping 1KB data from guest kernel, 2) dumping the process list and 3) dumping installed kernel module list. For both libvmi and secure VMI, we test the latency of each functionality a hundred times, and the average result is shown in Table 6. The time shown in the table does not include the initialization phase of libvmi, which consists of identifying guest and getting guest VM information. Compared with libvmi, VMI in COLONY can reduce about 75% time for dumping 1KB data from the guest. The performance improvement is mainly because that VMI in COLONY can directly access the memory mappings while libvmi needs to invoke the Domain-0 kernel as well as the hypervisor.

6.3.2 Secure Device Accessing

We use the Runtime Integrity Checker (RTIC) to evaluate the overhead of secure device accessing with COLONIES on

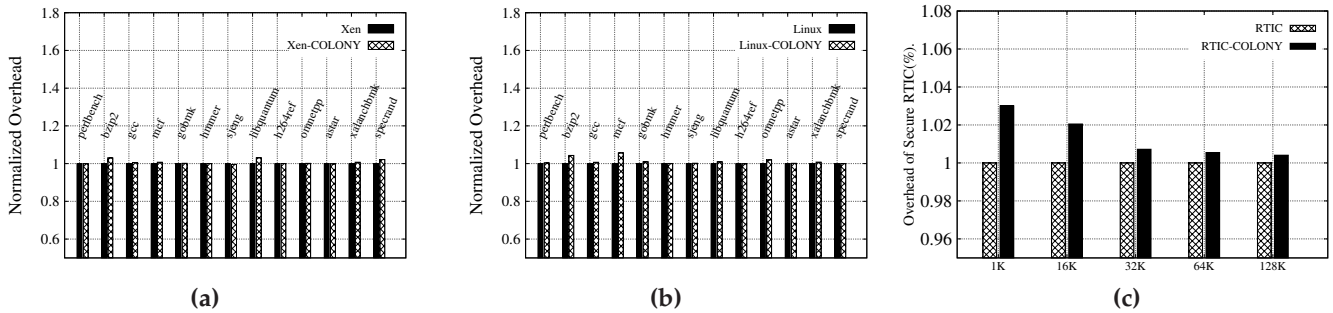


Fig. 5. Figure (a) and (b) show the normalized overhead of our system of SPEC_CPU 2006 INT applications on Xen and Linux, respectively. Figure (c) shows the overhead of *secure RTIC*. Lower is better.

TABLE 6
Latency of different VMI operations, in cycles.

Operation	libvmi	COLONY-VMI	Normalized
Dump symbol (1KB)	223,505	56,380	25.22%
Process list	2,118,113.5	1,532,319.5	72.34%
Module list	342,120	204,110.7	59.66%

Exynos board. *RTIC* is a commonly used security-related device which can calculate hash values of at most five different memory regions. It can be used to detect whether some memory regions have been tampered with. We leverage RTIC to perform SHA1 hashing on five memory regions with sizes from 1K to 128K. For each case, we test the latency a hundred times. Figure 5(c) shows the overhead of secure RTIC with COLONIES (*RTIC*-COLONY). It only incurs overhead from 0.5% to 3%.

6.3.3 Secure Syscall Tracer

The secure syscall tracer in COLONY runs as a system service. To evaluate its performance, we write a test program to trace all the syscalls invoked by a target process, based on the secure syscall tracer service. The functionality of the test program is similar to the *strace* application, which uses the *ptrace* provided by Linux kernel. We use both the secure tracer and the strace to trace the syscalls invoked by Nginx. After that, we use *ab* benchmark, with different numbers of threads, to measure the throughput of Nginx, both ab and Nginx server run in the HiKey board to bypass the influence of network. For each case, we use *ab* to send 10 thousand requests and calculate the average throughput. As Figure 6(a) shows, both the strace and our secure tracer cause a huge overhead (strace causes about 85% performance slowdown). Compared with strace, our secure tracer in COLONY has about 5% overhead on average.

6.4 Performance of TrustZone Virtualization

To answer *Question-3*, we perform a detailed evaluation about the performance of the TrustZone virtualization (vTZ) based on COLONY. Before the evaluation, we ported two widely used secure OSes, namely seL4 [6] and OP-TEE [4], to run inside the virtual secure world. First, vTZ leverages Xen's multi-boot loader to load secure OS kernel's image, so we need to add a multi-boot header in the image. Second, we add a new description file (e.g., platform_config.h in OP-TEE) to describe the memory layout of our guest secure world. Finally, since vTZ already provides a secure context switch, we remove the context switching logic in secure OS.

6.4.1 Micro-benchmark

We test the latency of critical operations in TrustZone and vTZ. We perform each operation a hundred times and calculate the average latency.

World Switch Overhead: For the physical TrustZone, the time of switching between two worlds is about 17,840 cycles on Exynos board and 1,294 cycles on HiKey board (shown in Table 7). The cost includes context saving and restoring in the schedule COLONIES. For the virtual TrustZone, one switching between guest's normal world and guest secure world is about 34,199 cycles on Exynos and 6,851 cycles on HiKey. The overhead is still acceptable since world switching happens rarely and thus has little effect on TrustZone-based applications.

Secure Configuration Overhead: A secure OS usually configures system resource partitioning during initialization or occasional run-time protection. Table 7 shows the overhead of these configurations with COLONY. The native value is performing configuration by hardware in the real secure world. Since HiSilicon, the vendor of hikey's SoC, does not publish the register mapping of *TZASC* or *TZPC*, the native time of HiKey is not provided. Same as world switching, secure configuration operations happen rarely, so the overhead will have limited effect on the real applications.

TABLE 7
Single operation overhead (unit: *cycle*).

	TrustZone (ARMv7)	vTZ (ARMv7)	TrustZone (ARMv8)	vTZ (ARMv8)
World Switching	17840	43199	1294	7732
Memory Partition	5798	15234	n/a	8155
Device Partition	1886	14311	n/a	7791
Interrupt Partition	1073	8943	755	6839

6.4.2 Application Overhead

Single Guest: We test four real applications (ccrypt, mcrypt, GnuPG and GoHttp) and compare them with original Xen on ARMv7 and ARMv8 platform. We use these applications to encrypt/transfer file about 1KB, and protect the encryption logic in real secure world/guest virtual secure world. The application/guest VM is pinned on one physical core in native environment/virtualization environment, respectively. We run each case 20 times and calculate the average execution time. Figure 6(b) and Figure 6(c) show the overhead. vTZ has little overhead compared with original Xen on both ARMv7 and ARMv8 platforms.

Multi-Guest: We compare the concurrent performance of vTZ with native environment, real TrustZone and original virtualization environment (Xen). The GoHttp server

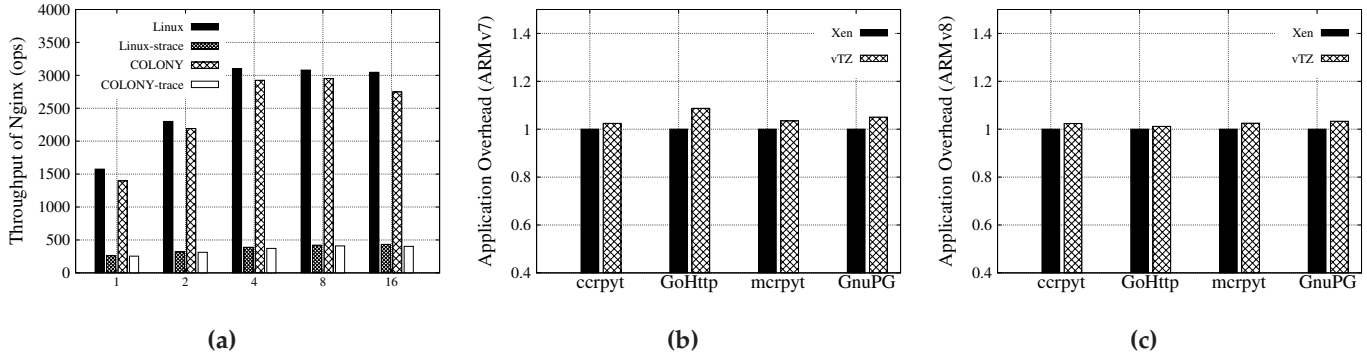


Fig. 6. Figure (a) shows the throughput of Nginx with the secure syscall tracer of our system (higher is better). Figure (b) and (c) show the application overhead of vTZ (lower is better).

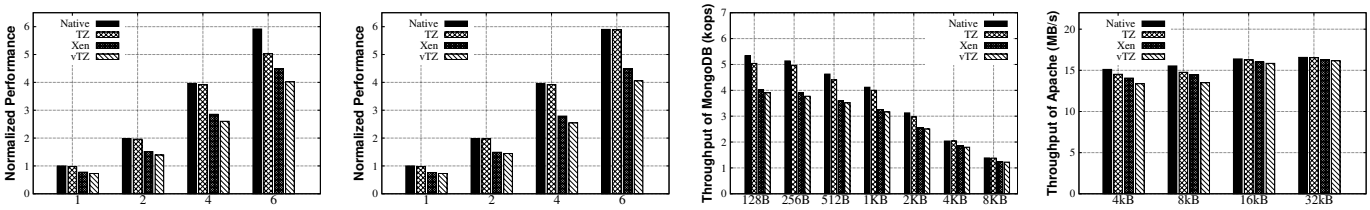


Fig. 7. Figure (a) and (b) show the throughput of GoHttp with different concurrencies. Figure (c) and (d) show the throughput of MongoDB and Apache, with different object sizes and TCP buffer sizes. Higher is better.

is used to do the evaluation, and we protect its encryption logic in secure world/guest virtual secure world. In native environment (including protection with TrustZone), each GoHttp server runs as a normal process. In virtualization environment (including protected by vTZ), each of them runs in one guest VM. The client, which sends the https request, runs in the same guest with the server to bypass the network overhead. We test the throughput of the data transmission between the client and the GoHttp server. To reduce the performance fluctuation, the client downloads a 20MB file and calculates a throughput. For each case, we run the test 20 times and calculate the average throughput. We only run at most 6 applications concurrently because the memory on the board limits the number of VMs. The results are shown in Figure 7 (a) and (b). vTZ has about a 5% performance slowdown compared with original Xen on ARMv8 implementation, and less than 30% performance slowdown compared with native environment.

Server Application Overhead: We also evaluate two widely used server applications, MongoDB [3] and Apache [1] on HiKey board. Same as the multi-guest evaluation, we run applications on four different environments. The clients are executed together with the server to bypass the network overhead. One difference is that the guest has eight virtual cores instead of one.

Figure 7 (c) shows the throughput of the insert operation of MongoDB. The client continually inserts 10 thousand objects to the server and calculates the throughput. We perform the evaluation with different sizes of objects. The result shows that vTZ has little overhead compared with the virtualization environment. For Apache (shown in Figure 7 (d)), we evaluate the downloading throughput by downloading a file (size is 100MB) from the server with https protocol. We run the test case 20 times and calculate the average throughput. The result shows that using virtual TrustZone causes less than 5% overhead in the virtualization environment.

7 Security Analysis and Discussion

7.1 Minimizing TCB

There are two kinds of TCB in our system: system TCB and service TCB. Components outside the system TCB cannot: 1) break the isolation between a COLONY with the untrusted system software or between different Colonies, and 2) provide fake system-level semantics to a COLONY. In our system, the secure monitor is the only system TCB. The size of the monitor itself is about 2K lines of C code (including inline assembly), and the size of the binary is about 12KB.

On the contrary, the service TCB depends on the semantic of service. Our system provides the capability control and the cross-user isolation, so that a service can be decoupled to multiple components, each running in one COLONY. It is known that by splitting a service into a number of small isolated, mutually untrusted components with explicit communication channels, the security could be enhanced [31]. The service TCB only contains components that may compromise the entire service. For example, if a service component sends malicious replies to other components but will not affect their correctness or will be detected/rejected, then it is not a part of the service TCB.

7.2 Attacking COLONIES

Code-reuse or Return-to-user/guest Attacks: Attackers may issue ROP (Return-Oriented Programming) attacks to ex-

ecute critical instructions (e.g., switching translation table) and bypass the monitor. ARM has several ISAs (e.g., AArch64, AArch32), the instructions are aligned. The monitor ensures that there is no critical instruction under any ISAs in the system software's text section. After that, it ensures that only the code of the system software can be mapped as executable in the privileged mode, thus return-to-user/guest attack can also be prevented.

DMA Attack: An attacker may try to access COLONY's memory or inject code into system software by leveraging Direct Memory Access (DMA). We defend against this attack by controlling System Memory Management Unit (SMMU), which performs address translation for DMA. SMMU is controlled by certain memory mapped registers. The monitor only maps these memory regions to secure world. By exclusively controlling the SMMU, the monitor ensures that no DMAs can access system software's text section or COLONY's memory.

Debugging Attack: The attacker may try to bypass the monitor by setting debug checkpoint before invoking it. Then the attacker can perform additional operations. Our system controls the entry points of all exception handlers, and the debug procedure is also under control. Thus, the debug point before invoking the monitor in the system software will trigger an infinite iteration, since the first instruction of the debug exception handler is invoking the monitor. This is a kind of DoS attack and is not considered in this paper.

Injecting Fake Exceptions: A malicious system software may try to inject fake exceptions. The monitor can detect them by checking exception registers. For example, on AArch64, the registers *ELR_ELx* and *SPSR_ELx* are set by hardware when an exception happens. The system software must set fake values to these registers to inject a fake exception. However, our system ensures that only the monitor can modify the two registers, as mentioned in Section 4.3, which will check and reject such fake values.

7.3 Security Limitations

Although COLONIES are isolated from each other, it is still possible that a compromised callee COLONY may attack the caller by providing a malicious return value, a.k.a., Iago attacks [20]. We rely on the service itself to defend these attacks by verifying the return value of the callee. Our design also does not consider hardware-based attacks, side-channel attacks and DoS attacks.

7.4 Combining with Non-Privileged TEE

Our system relies on existing non-privileged TEEs to protect clients. The COLONY abstraction is portable and can be combined with different non-privileged TEEs, including hardware TEE (such as SGX-like enclaves) and software TEE (such as TZ-Container [30]). Our current implementation requires a software TEE mechanism.

We provide a set of COLONY interfaces for a non-privileged TEE to interact with a COLONY. Since non-privileged TEEs often protect their CPU context and memory, to integrate them with our system, we need to allow the secure monitor to access such status, so that the non-privileged TEEs can pass arguments to the secure services located in a COLONY.

8 CONCLUSION

This paper introduces a new TEE design that can access system-level semantics and achieve better extensibility. We use the new TEE to protect four system-level services: virtual machine introspection (VMI), secure device accessing, syscall tracer, and TrustZone virtualization. We implemented the our design in both Linux kernel and Xen hypervisor. The evaluation shows that our system only causes small performance overhead.

9 ACKNOWLEDGMENTS

This work is supported in part by National Key Research and Development Program of China (No. 2020AAA0108502), China National Natural Science Foundation (No. 61972244, U19A2060, 61925206, 61732010).

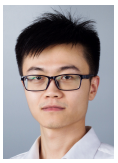
REFERENCES

[1] Apache http server. https://www.apache.org/, visited on 8 Feb 2021.
[2] Arm trusted firmware. https://github.com/ARM-software/arm-trusted-firmware, visited on 8 Feb 2021.
[3] Mongodb. https://www.mongodb.com/, visited on 8 Feb 2021.
[4] Optee. https://github.com/OP-TEE/, visited on 8 Feb 2021.
[5] Qualcomm security. https://www.qualcomm.com/media/documents/files/guard-your-data-with-the-qualcomm-snapdragon-mobile-platform.pdf, visited on 8 Feb 2021.
[6] sel4. http://sel4.systems, visited on 8 Feb 2021.
[7] Software guard extensions programming reference. https://software.intel.com/sites/default/files/managed/48/88/329298-002.pdf, visited on 8 Feb 2021.
[8] Trustonic inc. https://www.trustonic.com/, visited on 8 Feb 2021.
[9] Xen on arm. https://wiki.xen.org/wiki/Xen_ARM_with_Virtualization_Extensions, visited on 8 Feb 2021.
[10] T. Alves and D. Felton. Trustzone: Integrated hardware and software security. *ARM white paper*, 3(4):18–24, 2004.
[11] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'Keeffe, M. L. Stillwell, et al. Scone: Secure linux containers with intel sgx. In *USENIX Symposium on Operating Systems Design and Implementation*. USENIX Association, 2016.
[12] A. M. Azab, P. Ning, J. Shah, Q. Chen, R. Bhutkar, G. Ganesh, J. Ma, and W. Shen. Hypervision across worlds: Real-time kernel protection from the arm trustzone secure world. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 90–102. ACM, 2014.
[13] A. M. Azab, P. Ning, and X. Zhang. Sice: a hardware-level strongly isolated computing environment for x86 multi-core platforms. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 375–388. ACM, 2011.
[14] A. M. Azab, K. Swidowski, J. M. Bhutkar, W. Shen, R. Wang, and P. Ning. Skee: A lightweight secure kernel-level execution environment for arm. In *Network & Distributed System Security Symposium (NDSS)*, pages 21–24, 2016.
[15] A. Baumann, M. Peinado, and G. Hunt. Shielding applications from an untrusted cloud with haven. *ACM Transactions on Computer Systems (TOCS)*, 33(3):8, 2015.
[16] R. Boivie and P. Williams. Secureblue++: Cpu support for secure execution. *IBM Research Report, RC25287 (WAT1205-070)*, pages 1–9, 2012.
[17] F. Brasser, D. Gens, P. Jauernig, A.-R. Sadeghi, and E. Stapf. Sanctuary: Arming trustzone with user-space enclaves. pages 1–15, 2019.
[18] D. Cerdeira, N. Santos, P. Fonseca, and S. Pinto. Sok: Understanding the prevailing security vulnerabilities in trustzone-assisted tee systems. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P), San Francisco, CA, USA*, pages 18–20, 2020.
[19] D. Champagne and R. Lee. Scalable architectural support for trusted software. In *Proc. HPCA*, pages 1–12, 2010.
[20] S. Checkoway and H. Shacham. *Iago attacks: Why the system call api is a bad untrusted rpc interface*, volume 41. ACM, 2013.

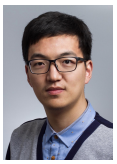
[21] X. Chen, T. Garfinkel, E. Lewis, P. Subrahmanyam, C. Waldspurger, D. Boneh, J. Dwoskin, and D. Ports. Overshadow: a virtualization-based approach to retrofitting protection in commodity operating systems. In *Proc. ASPLOS*, pages 2–13. ACM, 2008.
[22] S. Chhabra, B. Rogers, Y. Solihin, and M. Prvulovic. SecureME: A Hardware-Software Approach to Full System Security. In *ICS*, 2011.
[23] Y. Cho, J. Shin, D. Kwon, M. Ham, Y. Kim, and Y. Paek. Hardware-assisted on-demand hypervisor activation for efficient security critical code execution on mobile devices. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 565–578. USENIX Association, 2016.
[24] V. Costan, I. Lebedev, and S. Devadas. Sanctum: Minimal hardware extensions for strong software isolation. In *25th {USENIX} Security Symposium ({USENIX} Security 16)*, pages 857–874, 2016.
[25] J. Criswell, N. Dautenhahn, and V. Adve. Virtual ghost: Protecting applications from hostile operating systems. *ACM SIGARCH Computer Architecture News*, 42(1):81–96, 2014.
[26] N. Dautenhahn, T. Kasampalis, W. Dietz, J. Criswell, and V. Adve. Nested kernel: An operating system architecture for intra-kernel privilege separation. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 191–206. ACM, 2015.
[27] J. L. Henning. Spec cpu2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News*, 34(4):1–17, 2006.
[28] O. S. Hofmann, S. Kim, A. M. Dunn, M. Z. Lee, and E. Witchel. Inktag: secure applications on an untrusted operating system. *ACM SIGPLAN Notices*, 48(4):265–278, 2013.
[29] Z. Hua, J. Gu, Y. Xia, H. Chen, B. Zang, and H. Guan. vtz: Virtualizing arm trustzone. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 541–556. USENIX Association, 2017.
[30] Z. Hua, Y. Yu, J. Gu, Y. Xia, H. Chen, and B. Zang. Tz-container: Protecting container from untrusted os with arm trustzone. *SCIENCE CHINA Information Sciences*, pages 1–16, 2020.
[31] T. Hunt, Z. Jia, V. Miller, H. Tyler, J. Zhipeng, M. Vance, C. J. Rossbach, and E. W. Witchel. Isolation and beyond: Challenges for system security, 2019.
[32] J. Jang, S. Kong, M. Kim, D. Kim, and B. B. Kang. Secret: Secure channel between rich execution environment and trusted execution environment. In *NDSS*, pages 180–195, 2015.
[33] Y. Kwon, A. M. Dunn, M. Z. Lee, O. S. Hofmann, Y. Xu, and E. Witchel. Sego: Pervasive trusted metadata for efficiently verified untrusted system services. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 277–290. ACM, 2016.
[34] W. Li, H. Li, H. Chen, and Y. Xia. Adattester: Secure online mobile advertisement attestation using trustzone. In *MobiSys*, 2015.
[35] W. Li, M. Ma, J. Han, Y. Xia, B. Zang, C.-K. Chu, and T. Li. Building trusted path on untrusted device drivers for mobile devices. In *Proceedings of 5th Asia-Pacific Workshop on Systems*, page 8. ACM, 2014.
[36] H. Liang, M. Li, Y. Chen, L. Jiang, Z. Xie, and T. Yang. Establishing trusted i/o paths for sgx client systems with aurora. *IEEE Transactions on Information Forensics and Security*, 15:1589–1600, 2019.
[37] L. W. McVoy, C. Staelin, et al. lmbench: Portable tools for performance analysis. In *USENIX annual technical conference*, pages 279–294. San Diego, CA, USA, 1996.
[38] B. Payne. Libvmi introduction: Vmitools, an introduction to libvmi, 2014.
[39] N. Santos, R. Rodrigues, K. P. Gummadi, and S. Saroiu. Policy-sealed data: A new abstraction for building trusted cloud services. In *Usenix Security*, 2012.
[40] A. SEV-SNP. Strengthening vm isolation with integrity protection and more. *White Paper, January*, 2020.
[41] H. Sun, K. Sun, Y. Wang, and J. Jing. Trustotp: Transforming smartphones into secure one-time password tokens. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 976–988. ACM, 2015.
[42] H. Sun, K. Sun, Y. Wang, J. Jing, and H. Wang. Trustice: Hardware-assisted isolated computing environments on mobile devices. In *Dependable Systems and Networks (DSN), 2015 45th Annual IEEE/IFIP International Conference on*, pages 367–378. IEEE, 2015.
[43] C.-C. Tsai, D. E. Porter, and M. Vij. Graphene-sgx: A practical library os for unmodified applications on sgx. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, page 8, 2017.
[44] J. Yang and K. G. Shin. Using hypervisor to provide data secrecy for user applications on a per-page basis. In *Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 71–80. ACM, 2008.
[45] F. Zhang, J. Chen, H. Chen, and B. Zang. Cloudvisor: retrofitting protection of virtual machines in multi-tenant cloud with nested virtualization. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 203–216. ACM, 2011.

Yubin Xia received his Ph.D. degree in computer science from Peking University in 2010. He is currently an associate professor of the Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University. He is a member of ACM and IEEE. His research interests include system software, computer architecture and system security.

Zhichao Hua received his PH.D degree in computer science from Shanghai Jiao Tong University in 2020. He is currently an assistant professor of the Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University. His research interests include operating system, virtualization and system security.



Yang Yu received his Ph.D. degree from School of Computer Science, Fudan University in 2016. He is now a senior researcher in Shanghai Gejing Information Technology Co., Ltd. His research interests include computer architecture, system security and Java virtual machine.



Jinyu Gu received the BS degree in software engineering in 2016, from Shanghai Jiao Tong University. He is now a Ph.D student at the Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University. His research interests include operating system and system security.

Haibo Chen received his Ph.D. degree in computer science from Fudan University in 2009. He is currently a professor at the Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University. He is a distinguished member of ACM and senior member of IEEE. His research interests are in operating systems and parallel and distributed systems.

Binyu Zang received his Ph.D. degree in computer science from Fudan University in 1999. He is currently a professor at the Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University. He is a member of ACM and IEEE. His research interests include compilers, computer architecture and systems software.

Haibing Guan obtained his Ph.D. degree from Tongji University in 1999 and worked in Shanghai Jiao Tong University since 2002. He is now a professor of Shanghai Jiao Tong University. He is a member of ACM and IEEE. His major research interests include computer system, cloud computing.